

Java Dependency Management with Ivy

Part I – The Basics

Java Dependencies

Along with the rich enterprise libraries that come as part of the language, one of the biggest advantages of Java is the vast number of third party libraries available. For example if you are writing an enterprise web application, GWT [1], Spring [2] and Hibernate [3] provide a framework with a huge amount of pre-existing functionality.

The size and number of dependencies grows as your application grows. GWT and Spring alone, without their dependencies, are more than 7MB. The ideal place to put dependencies is in a source control repository as part of your project so that when you or your continuous integration server check out the project for the first time all the dependencies are there. Then you don't have to go and get them and store them locally in a location that is agreed by the entire development team.

Storing the dependencies for a single project in a source control repository isn't too bad, provided there is plenty of room on the source control server.. However, if you have more than one project using the same or similar sets of dependencies the amount of space taken up in the source control repository starts to get a bit ridiculous. And then when a new version of a library comes out and you upgrade, even more space is wasted as the differences between binary jars cannot be detected, so the entire jar must be replaced.

Solutions to Dependency Storage

There are a number of ways to solve the problem of dependencies taking up too much space in a source control repository.

You could check your dependencies into a single place in the repository, which does mean they're still taking up *some* space, and then use a link (e.g. svn externals [4]) to each project so that when the project is checked out the dependencies are checked out as well. Although this would work well it can be a pain to set-up and maintain, especially if you link to individual jars.

Maven [5] allows you to specify which dependencies you need and automatically downloads them for you when you check your project out. This means you do not have to check your dependencies into your source control system, just maintain a configuration file. The drawbacks are that you're restricted to the third party libraries available via Maven and you have to buy into the whole Maven project layout and configuration.

Ivy [6] provides the dependency management features of Maven, without the need to buy into any particular project layout. The default locations to download dependencies from are the Maven repositories, but you can also specify other locations and set-up your own own local repositories. Ivy creates a local cache and retrieves dependencies from it every time a project is checked out or another dependency is added.

Ivy represents the best solution as you don't need to store any dependencies in your source control system and you're not restricted to a particular project layout. In this article I am going to describe how to use Ivy to manage dependencies for a Java project using Ant [7] and the IvyDE [8] plugin for Eclipse [9].

Using Ivy with Ant

To use Ivy with Ant download the latest release of Ivy from the Ivy website, extract the Ivy jar file (at time of writing: `ivy-2.1.0-rc2.jar`) and either:

- copy it to Ant's lib directory (`ANT_HOME\lib`)
- copy it to another directory and use an Ant `taskdef` task to reference it:

```
<path id="ivy.lib" >
  <pathelement location = "thirdparty\ivy\ivy-2.1.0-rc2.jar"/>
</path>

<taskdef uri="antlib:fr.jayasoft.ivy.ant"
  resource="fr/jayasoft/ivy/ant/antlib.xml"
  classpathref="ivy.lib"/>
```

The easiest way to demonstrate using Ivy with Ant is with a simple application that has a single dependency:

```
import org.apache.commons.lang.WordUtils;

public class IvyAnt
{
    public static void main(String[] args)
    {
        final String msg = "hello, world!";
        System.out.println(WordUtils.capitalizeFully(msg));
    }
}
```

This application uses the `org.apache.commons.lang` library to correctly title-case a simple message. The basic Ant script (`build.xml`) for an application like this is as follows:

```
<project name = "IvyAnt" basedir="." default = "compile" >
  <property name = "src.dir" value="${basedir}/src" />
  <property name = "bin.dir" value="${basedir}/bin" />

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
  </target>

  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}" destdir="${bin.dir}" fork="true">
      </javac>
    </target>
</project>
```

It only does two things:

1. Creates a `bin` folder for the compiled class files.
2. Attempts to compile java files into class files.

If run from the command line it fails as it cannot find the jar file for `apache.commons.lang`. Enter Ivy!

To get Ivy to download and cache dependencies for us, we need to tell it where to get them from. This is the purpose of the `ivy.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">

  <info organisation="purpletube.net" module="IvyAnt" status="integration"/>
  <dependencies>
    <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
  </dependencies>
</ivy-module>
```

The Ivy Quick Start Guide [10] describes the format of the `ivy.xml` very well, but in summary the `info` tag is used to give information about the module for which we are defining dependencies. In the dependencies section, the `org` and `name` attributes define the organization and module name of the dependency. The `rev` attribute is used to specify the revision of the module you depend on. Dependencies can be looked up in the Maven repository [11]. Once found, the Maven POM (Project Object Model - an XML representation of a Maven project held in a file), for example:

```
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.0</version>
</dependency>
```

can be converted to an Ivy dependency. Use the `groupId` as organization, the `artifactId` as name, and the `version` as `rev`.

Put the `ivy.xml` file in the same directory as your Ant `build.xml`. To get Ant to use Ivy and resolve dependencies the Ivy namespace and the Ivy retrieve task need to be added:

```
<project name = "IvyAnt" basedir="." default = "compile"
  xmlns:ivy="antlib:org.apache.ivy.ant">

  <property name = "src.dir" value="${basedir}/src" />
  <property name = "bin.dir" value="${basedir}/bin" />

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
    <ivy:retrieve />
```

```

</target>

<target name = "compile" depends= "init">
  <javac srcdir="${src.dir}" destdir="${bin.dir}" fork="true">
  </javac>
</target>
</project>

```

Running `build.xml` again will download and cache `apache.commons.lang` and its dependencies, but the build will still fail. Ignore the warning about a missing Ivy settings file for the time being. The settings file is only used if you want to use settings other than the default. The rest of the message is telling you that Ivy is downloading the required libraries and caching them. The default cache location is a directory called `.ivy2/cache` below the user's home directory. The location Ivy uses as the user's home can be changed by setting the `ivy.default.ivy.user.dir` property in `build.xml` file.

You should also see that a new directory called `lib` has been created, which contains the `apache.commons.lang` jar files. The location and name of this directory can be changed by setting the `ivy.lib.dir` Ant property. Even if you use the default `ivy.lib.dir` value, you must declare it as a property in `build.xml` so that the Java compiler knows where to find the jars.

Even though Ivy has downloaded and cached the dependency and even copied it to a local location within the project, the Java compiler needs to know where to look for the jar files. This is achieved by setting up a path id and using it as the Java compiler's class path:

```

<project name = "IvyAnt" basedir="." default = "compile"
  xmlns:ivy="antlib:org.apache.ivy.ant">
  <property name = "src.dir" value="${basedir}/src" />
  <property name = "bin.dir" value="${basedir}/bin" />
  <property name = "ivy.lib.dir" value="${basedir}/lib" />

  <path id="lib.path.id">
    <fileset dir="${ivy.lib.dir}" />
  </path>

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
    <ivy:retrieve />
  </target>

  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}" destdir="${bin.dir}" fork="true">
      <classpath refid = "lib.path.id"/>
    </javac>
  </target>
</project>

```

When you run the Ant script again you will see that Ivy recognises that it already has the dependencies cached, so it does not download them again.

Writing the Ant task to run the application is very simple also. All you need is a path id so that `java` can find the compiled class files and jar files and a `java` task:

```

<project name = "IvyAnt" basedir="." default = "run"
  xmlns:ivy="antlib:org.apache.ivy.ant">
    <property name = "src.dir" value="{basedir}/src" />
    <property name = "bin.dir" value="{basedir}/bin" />
    <property name = "ivy.lib.dir" value="{basedir}\lib" />

    <path id="lib.path.id">
      <fileset dir="{ivy.lib.dir}" />
    </path>

    <path id="run.path.id">
      <path refid="lib.path.id" />
      <path location="{bin.dir}" />
    </path>

    <target name = "init">
      <mkdir dir = "{bin.dir}"/>
      <ivy:retrieve />
    </target>

    <target name = "compile" depends= "init">
      <javac srcdir="{src.dir}" destdir="{bin.dir}" fork="true">
        <classpath refid = "lib.path.id"/>
      </javac>
    </target>

    <target name = "run" depends= "compile">
      <java classpathref="run.path.id" classname="IvyAnt"/>
    </target>
</project>

```

Running Ant one last time will give the output “Hello, World!”.

As your application grows all you need to do is search for the dependencies you need in mvnrepository.com, add them to `ivy.xml` and run Ant to retrieve them. Some dependencies, such as the Microsoft SQL Server driver for Java are not in the Maven repository and it is advantageous to set-up a local repository to host these types of dependencies. I'll describe how to do this in a later article.

It's worth making sure you are clear on the difference between a cache and a repository. A cache is usually local. When you do a build, Ivy checks the cache to see if you already have the required dependencies. If you do, it uses them, otherwise it looks in the repository for them and downloads them. Repositories can be local, but tend to be remote on the internet or on a central server in an organisation. Maven is a repository and stores a large number of libraries.

Using Ivy with Eclipse

Ivy is great if you're using Ant and the command line, but what if you do most of your development in Eclipse and you do not want to use Ant? IvyDe is a new and relatively immature plugin. It will resolve and add the dependencies declared in an `ivy.xml` file to the classpath of your Eclipse project. It is also an editor for `ivy.xml` files with auto

completion. The download instructions are the same as for most Eclipse plugins and are given on the IvyDE website [12].

As with Ivy itself, the easiest way to demonstrate the use of IvyDE with Eclipse is with a simple application that has a single dependency:

```
import org.apache.commons.lang.WordUtils;

public class IvyEclipse
{
    public static void main(String[] args)
    {
        final String msg = "hello, world!";
        System.out.println(WordUtils.capitalizeFully(msg));
    }
}
```

If you create a Java project called `IvyEclipse` and add the class above, Eclipse should complain that it cannot resolve the `import org.apache` or `WordUtils`. The solution is to add an `ivy.xml` file. To do this:

1. Right click on the project and go to New->Other
2. Expand IvyDE and select Ivy File.
3. Click Next
4. Click the Browser button next to the container edit box and select the project.
5. Enter an organisation (e.g. Purple Tube) and use the project name as the Module.
6. Click finish.

You should be presented with the following `ivy.xml` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ivy-module version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
  <info
    organisation="Purple Tube"
    module="IvyAnt"
    status="integration">
  </info>
</ivy-module>
```

Add the dependencies from the `ivy.xml` in the Ant example above and save:

```
<ivy-module version="1.0">
  <info organisation="Purple Tube" module="IvyEclipse" status="integration"/>
  <dependencies>
    <dependency org="commons-lang"
      name="commons-lang"
      rev="2.0"
      conf="default"/>
  </dependencies>
</ivy-module>
```

To get Ivy to resolve the dependencies, right click on the `ivy.xml` file and select Add Ivy Library. Just click Finish on the IvyDE Managed Libraries dialog box for the time being. Then a new element will appear in the project tree called "`ivy.xml [*]`" and the project errors will be resolved. Expand the element to see that `org.apache.common.lang` is being referenced in the Ivy cache.

Unlike Ivy, IvyDE does not copy the dependencies to the project. If you require this behaviour, for example if you need to copy dependent jar files to a web application's WAR directory, you need to use IvyDE in conjunction with Ivy and Ant and set the `ivy.lib.dir` property appropriately. Ivy Ant tasks and IvyDE will happily share the same `ivy.xml`.

When you modify `ivy.xml` and save it or check the project out of a source control system, IvyDE should resolve automatically. However, if it does not, right click on "`ivy.xml [*]`" and select Resolve.

Conclusion

Ivy itself is a well featured, immediately useful tool. It has allowed my team to significantly reduce the amount of space used in our subversion repository, while remaining in complete control of our dependencies. In this article I have only scratched the surface of what it can do. I am intending to explore its capabilities further in future articles.

IvyDE still feels very new and needs some work. It is mostly a convenience, replacing the need to run ant to resolve dependencies. I am hoping it's going to come along soon. For now I am tempted to start trying to add features myself.

References

- [1] Google Web Tool Kit:
- [2] Spring Framework: <http://www.springsource.org/>
- [3] Hibernate Object Resource Mapper: <https://www.hibernate.org/>
- [4] Svn Externals: <http://svnbook.red-bean.com/en/1.0/ch07s03.html>
- [5] Maven Software Project Management: <http://maven.apache.org/>
- [6] Ivy, The Agile Dependency Manager: <http://ant.apache.org/ivy/>
- [7] Ant: <http://ant.apache.org/>
- [8] Ivy Eclipse Plugin: <http://ant.apache.org/ivy/ivyde/>
- [9] Eclipse IDE: <http://www.eclipse.org/>

[10] Ivy Quick Start Guide: <http://ant.apache.org/ivy/history/2.1.0-rc1/tutorial/start.html>

[11] Maven Repository: <http://mvnrepository.com/>

[12] IvyDE Download Instructions: <http://ant.apache.org/ivy/ivyde/download.cgi>